

# Big O Notation






Time complexity of an algorithm

"How much time it takes to run a function as the size of the input grows."

Runtime

array  
number of elements

Const

array1 = [ , , , ,  ] n=5

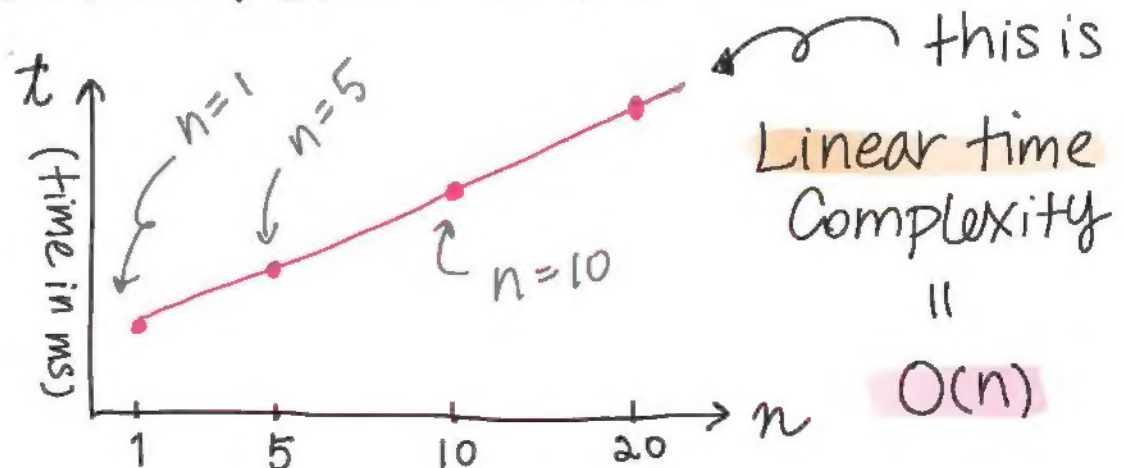
Let's see if there is a needle in the haystack!

```
JS Const numNeedles = (haystack, needle) => {  
  let count = 0  
  for (let i = 0; haystack.length; i++) {  
    if (haystack[i] === needle) count += 1;  
  }  
  return count;  
}
```



How long does it take to execute when the number of elements (n) is:

execution time grows linearly as array size increases!



# Big O Notation

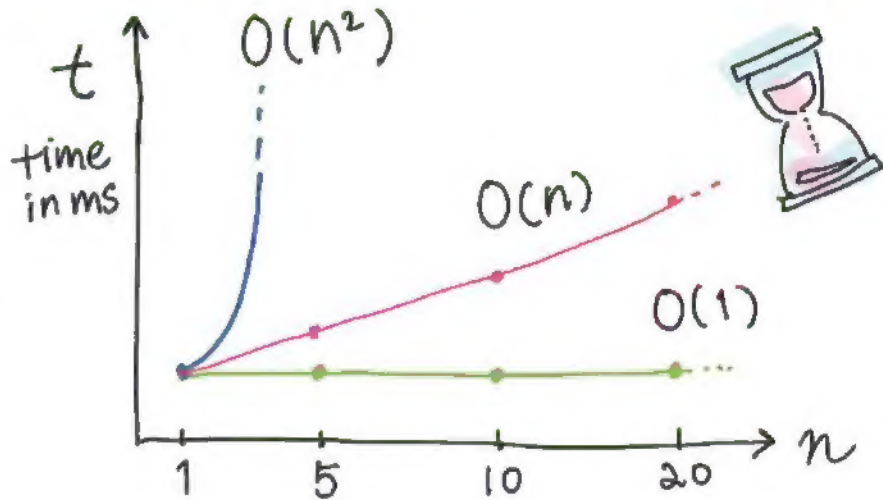
**JS** Let's see if we have some function that doesn't actually loop the array:

```
const alwaysTrueNoMatterWhat = (haystack) => {
  return true;
}
```

$n=5$   
 $n=10$   
 $n=20$   
 $\vdots$

↖ Array size has no effect on the runtime

☆ Constant time  
 $O(1)$



☆ Quadratic time =  $O(n^2)$

Const

array2 = [ , , , ,  ] ;

↖  $n=5$ , however the runtime is proportional to  $n^2$

**JS** Const hasDuplicates = (arr) => {

```

  for (let i = 0; i < arr.length; i++)
    let item = arr[i];
    if (arr.slice(i+1).indexOf(item) !== -1) {
      return true;
    }
  return false;
}
```

↖ ① Loop thru the array

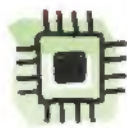
↖ ② Another array lookup w/ indexOf method



# Data Structures

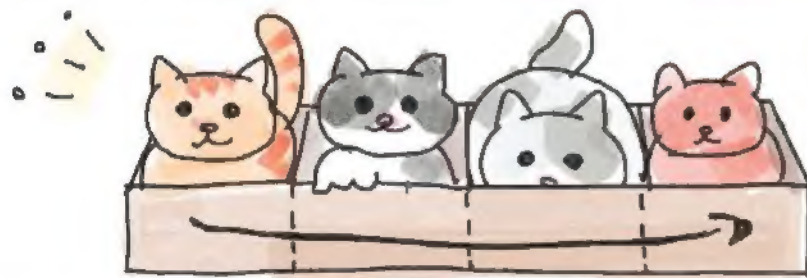
## Array & Linked List

**Array** a linear data structure, stored in contiguous memory locations.



memory

Address 200 204 208 212



Array [0] [1] [2] [3]

$a[0]$   
 $a[1]$   
 $a[2]$   
 $a[3]$

196

200

204

208

212

216

220

224

228



- ♥ Assume each 🐱 is an integer  
= requires 4 bytes space
- ♥ The array of 🐱 must be allocated contiguously!  
→ address 200 — 216

🎉 yay!

- ♥ can randomly access w/ index  
 $a[2] \rightarrow$  🐱
- ♥ contiguous = no extra memory allocated = no memory overflow

👎 meh!

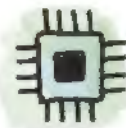
- 💀 fixed size. Large space may not be avail for big array  
= 🐱 took the space! =
- 💀 Insert & delete elements are costly.  
→ may need to create a new copy of the array & allocate at a new address.



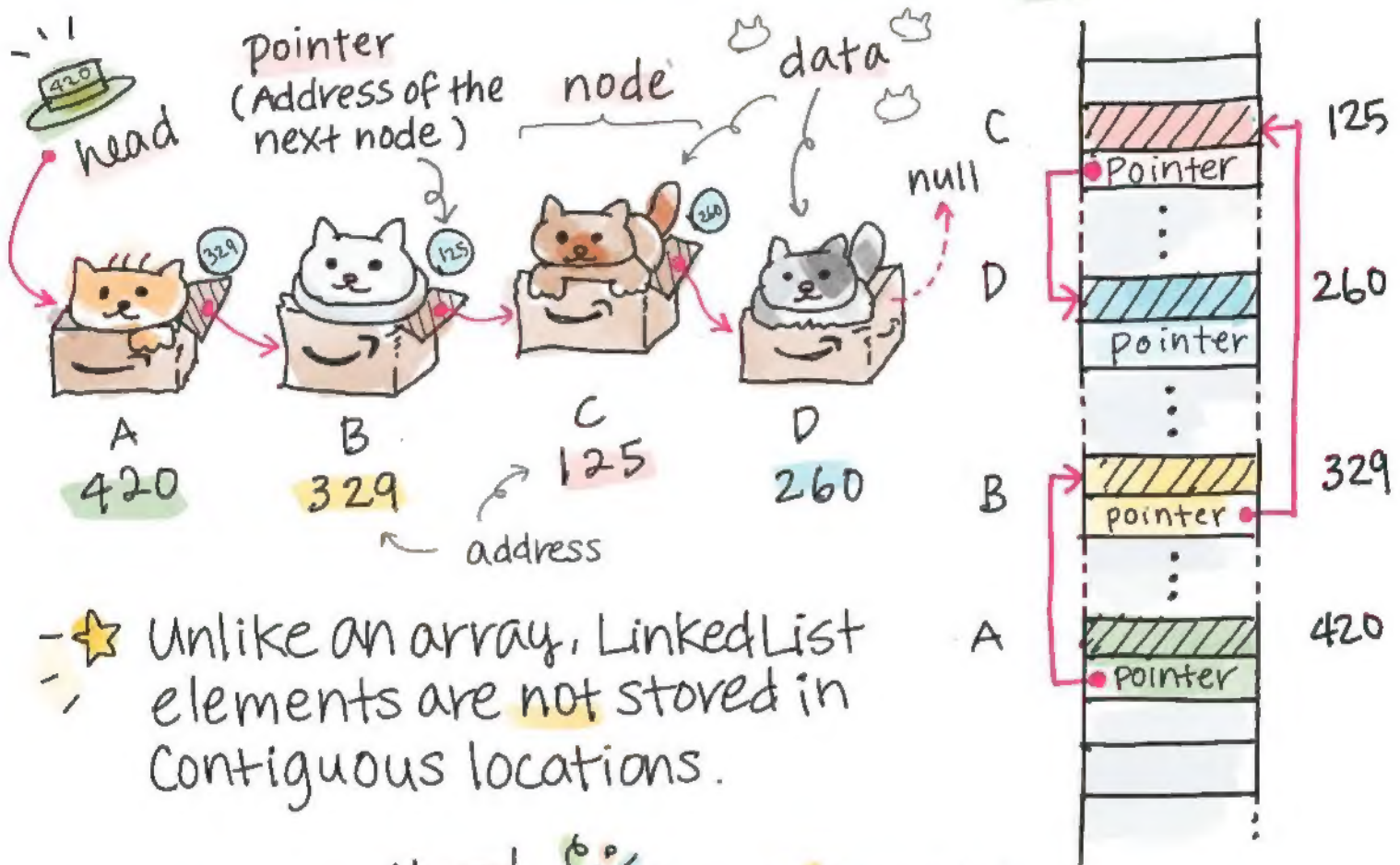
# Linked list

## Array & Linked List

- ★ a linear data structure
- ★ each element is a separated object & elements are linked w/ pointers



memory



- ★ Unlike an array, LinkedList elements are not stored in Contiguous locations.

Yay! 🎉

- ♥ Dynamic data  
= size can grow or shrink
- ♥ Insert & delete element are flexible.  
→ no need to shift nodes like array insertion
- ♥ memory is allocated at runtime

👎 meh!

- ☹ No random access memory.  
→ Need to traverse  $n$  times  
→ time complexity is  $O(n)$ . array is  $O(1)$
- ☹ Reverse traverse is hard



# Data Structures

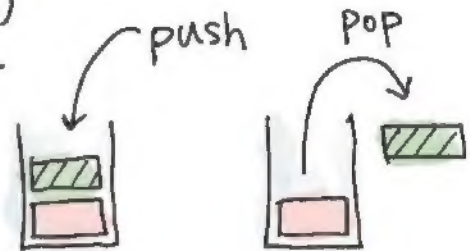
## Stack & Queue

FIFO

LIFO

@girlie\_mac

A stack is a LIFO (Last-in-First-out) data structure, where an element added last (=push) gets removed first (=pop)



♡ just like a stack of ice cream scoops!



☆ Stack as an array in JS

omg, the bottom one is always melting !!

arrays in JavaScript are dynamic!

```
let stack = [ ];  
stack.push('mint choc');  
stack.push('vanilla');  
stack.push('strawberry');  
let eaten = stack.pop();
```

stack is:  
['mint choc', 'vanilla', 'strawberry']  
eaten is 'strawberry'  
['mint choc', 'vanilla']

♡ Time complexity is  $O(1)$  for both pop + push.



# Data Structures

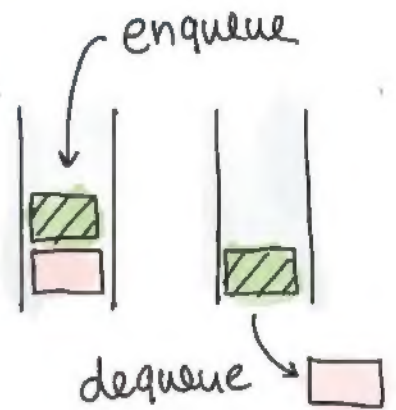
## Stack & Queue

FIFO

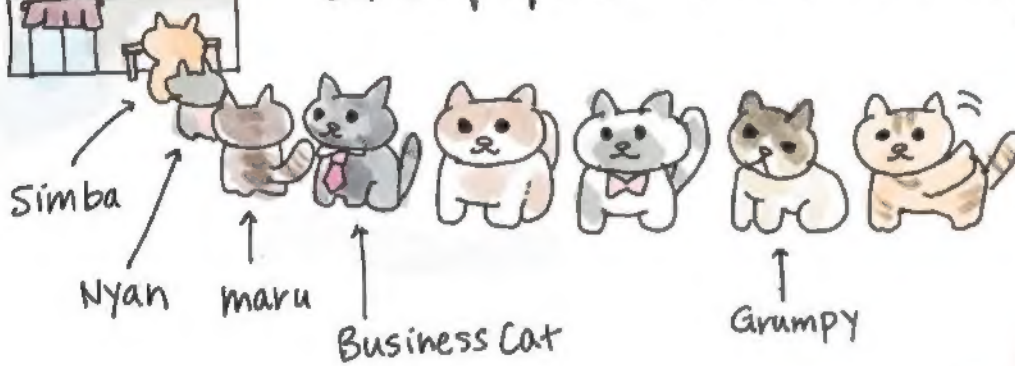
LIFO

@girlie\_mac

A queue is a FIFO (First-in-First-out) data structure, where an element added first (= enqueue) gets removed first (= dequeue)



just like waiting in line at a popular restaurant!



★ Stack as an array in JS

```
let queue = [ ];  
queue.push('Simba'); // ['Simba']  
queue.push('Nyan'); // ['Simba', 'nyan']  
queue.push('maru'); // ['Simba', 'nyan', 'maru']  
let eater = queue.shift(); // eater is 'Simba'
```

Time complexity should be  $O(1)$  for both enqueue + dequeue but JS `shift()` is slower!

if you queue.unshift('bad Kitty'), instead of push(), then the cat cuts in to the front of line!

Wrong!



# Data Structures Hash Table

- ♥ A hash table is used to index large amount of data
- ♥ Quick key-value look up.  $O(1)$  on average
  - ↳ Faster than brute-force linear search

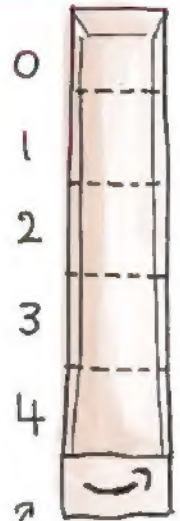
① Let's create an array of size 5.

We're going to add 🐱 data.

key = "Tabby"

value = "pizza"

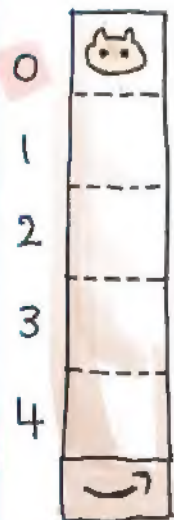
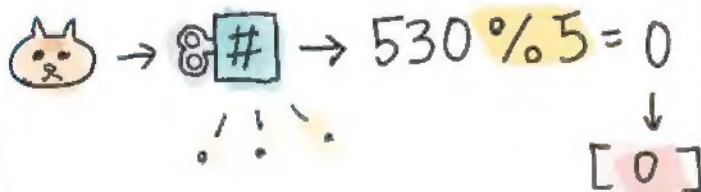
some data  
Let's say, favorite food!



② Calculate the hash value by using the key, "Tabby".  
e.g. ASCII code, MD5, SHA1

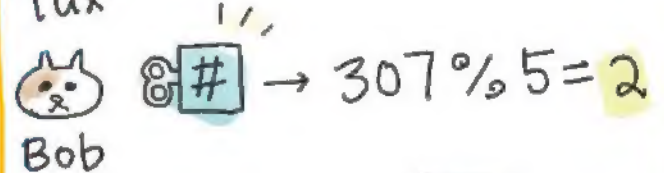
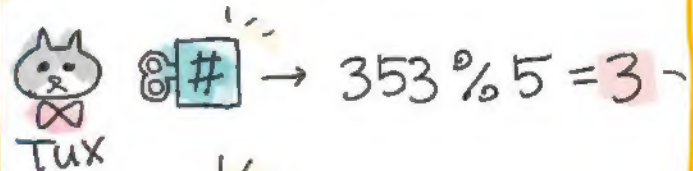


③ Use modulo to pick a position in the array!

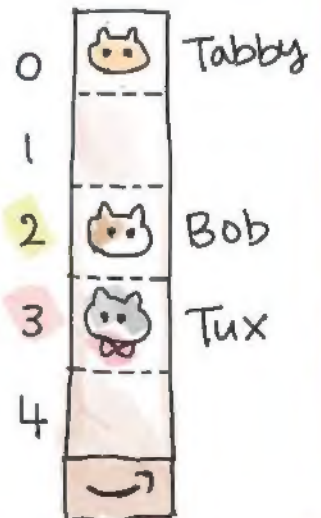


☆ The hash is divided by the size of the array.  
The remainder is the position!

④ Let's add more data.



Use the same method to add more 🐱





## ★ Collision!

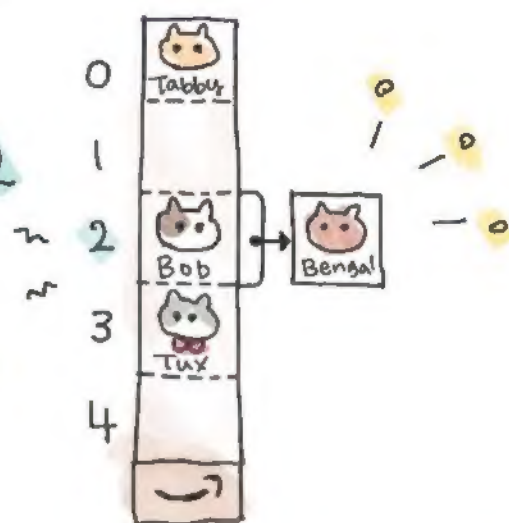
# Hash Table

@girlie-mac

Now we want to add more data.  
Let's add "Bengal".

🐱 "Bengal" →  $617\%5 = 2$

But [2] slot has been taken  
by "Bob" already! = collision!  
so let's chain Bengal next  
to Bob! = chaining

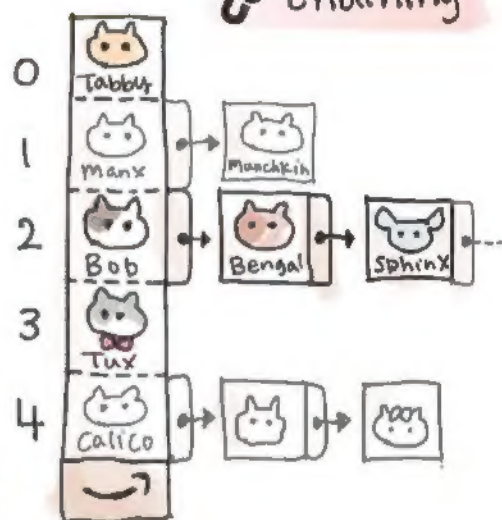


key: "Bengal"  
Value: "Dosa"

"Sphinx"  
"Fish +  
Chips"

Keep  
adding  
data

🔗 chaining



## 🔍 Searching for data

★ Let's look up the value for "Bob"

① Get the hash → 307

② Get the index →  $307\%5 = 2$

③ Look up Array [2] → found!

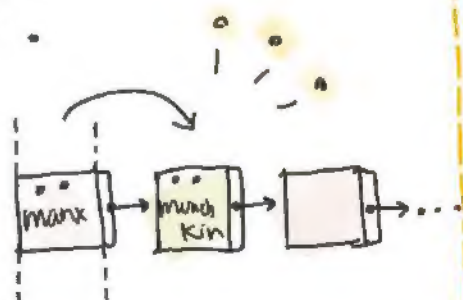
★ Let's look up "munchkin"

① Hash → 861

② Index →  $861\%5 = 1$

③ Array [1] → "Manx"

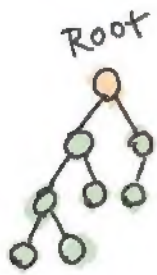
④ Operate a linear-search to find "munchkin"  
↳ Average  $O(n)$





# Data Structures

# Binary Heap



**Binary tree**

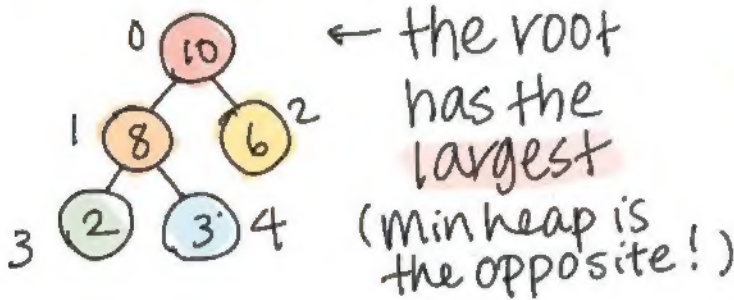
tree data structure  
each node has at most 2 children

Binary search tree

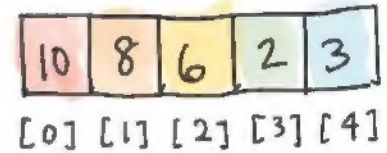
Binary heap

- Complete tree
- Min heap or max heap
- used for priority queue, heap sort etc.

## ★ Max heap

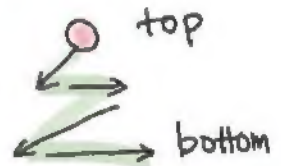


in array



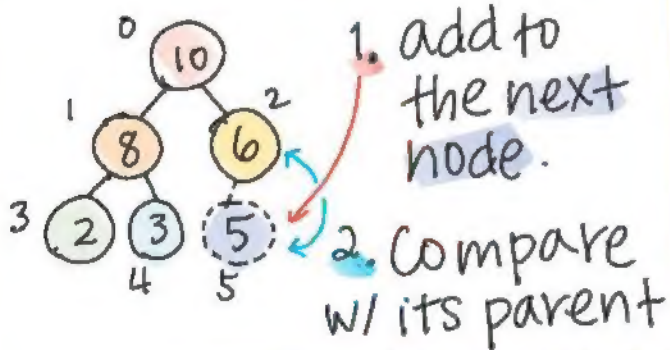
each node has 0 - 2 children

always fill top → bottom, left → right

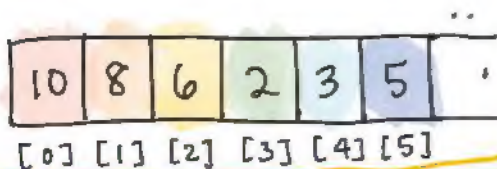


## ★ Insertion

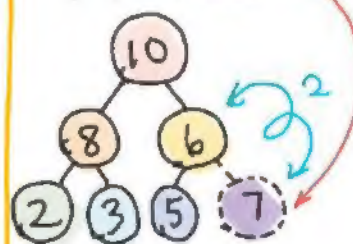
Let's add 5 to the heap!



1. add to the next node.  
2. Compare w/ its parent  
3. the parent is greater.  
Cool, it's done!  
Let's add more!



Add 7

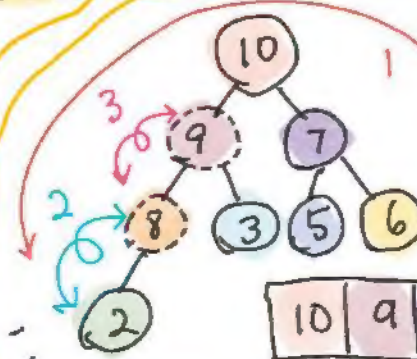
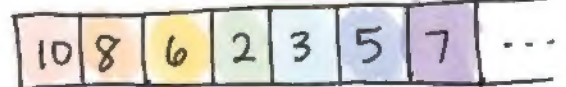


1. Add to the next node

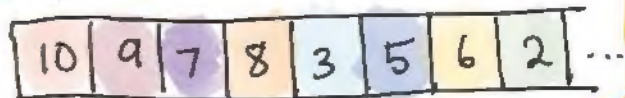
2. Compare w/ parent.

Oh, no!

the parent is smaller than its child! Swap them !!!

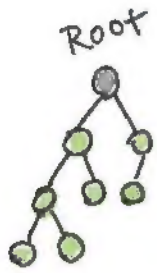


Add to the next node & repeat the process!





# Binary Search Tree



Binary tree

tree data structure  
each node has at most 2 children

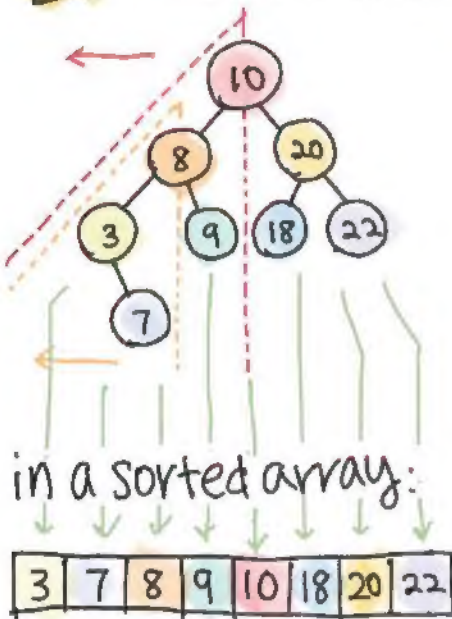
Binary heap

Binary Search Tree

a.k.a. Ordered or sorted binary tree

fast look up  
e.g. phone number lookup table by name

👍 Rule of thumb



★ each value of all nodes in the left subtrees is lesser

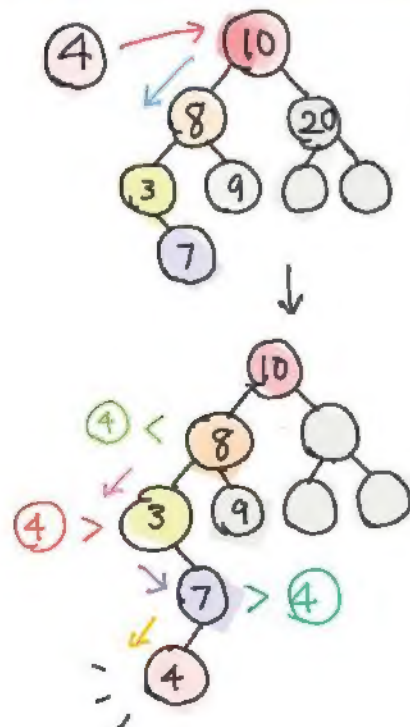
△ 10's left subtrees: 8, 3, 9, 7

△ 8: 3, 7 ← smaller than parent

★ each value of all nodes in the right subtrees is larger

★ no duplicate values

★ Insertion → Always add to the lowest spot to be a leaf! No rearrange!



Let's add 4

1. Compare w/ the root first.

2. 4 < 10 so go left.

3. then compare w/ the next, 8

4. 4 < 8 so go left

5. Compare w/ the 3

6. 4 > 3 so go right.

7. Compare w/ the 7

8. 4 < 7, so add to the left! Done.

Complexity:

Ave.  $O(\log n)$

Worst.  $O(n)$



**Refer link for more**